

CS 601.471/671 NLP: Self-supervised Models

Homework 4: Neural Language Modeling + Fixed-Window LMs

For homework deadline and collaboration policy, check the calendar on the course website¹

Name: Jacob Cunningham

Collaborators, if any: _____

Sources used for your homework, if any: _____

This assignment focuses on language modeling, while continuing to build up on our prior knowledge of neural networks. We will review several aspects about training neural nets and also extend it to modeling sequences in language.

Homework goals: After completing this homework, you should be comfortable with:

- thinking more deeply about training neural networks; debugging your neural network
- getting more engaged in using PyTorch for training NNs
- training your first neural LM

Concepts, intuitions and big picture

Multiple-choice questions.

1. Select the sentence that best describes the terms “model”, “architecture”, and “weights”.
 - Model and weights are the same; they form a succession of mathematical functions to build an architecture.
 - **An architecture is a succession of mathematical functions to build a model and its weights are those functions parameters.**
2. During forward propagation, in the forward function for a layer l you need to know what is the activation function in a layer (Sigmoid, tanh, ReLU, etc.). During Backpropagation, the corresponding backward function does not need to know the activation function for layer l since the gradient does not depend on it.
 - True

¹ <https://self-supervised.cs.jhu.edu/sp2024/>

- **False**

3. You have built a network using the tanh activation for all the hidden units. You initialize the weights to relative large values, using `randn(...,...)*1000`. What will happen?
 - It doesn't matter. So long as you initialize the weights randomly gradient descent is not affected by whether the weights are large or small.
 - This will cause the inputs of the tanh to also be very large, thus causing gradients to also become large. You therefore have to set the learning rate to be very small to prevent divergence; this will slow down learning.
 - This will cause the inputs of the tanh to also be very large, causing the units to be "highly activated" and thus speed up learning compared to if the weights had to start from small values.
 - **This will cause the inputs of the tanh to also be very large, thus causing gradients to be close to zero. The optimization algorithm will thus become slow.**
4. What is the "cache" used for in our implementation of forward propagation and backward propagation?
 - It is used to cache the intermediate values of the cost function during training.
 - **We use it to pass variables computed during forward propagation to the corresponding backward propagation step. It contains useful values for backward propagation to compute derivatives.**
 - It is used to keep track of the hyperparameters that we are searching over, to speed up computation.
 - We use it to pass variables computed during backward propagation to the corresponding forward propagation step. It contains useful values for forward propagation to compute activations.
5. Among the following, which ones are "hyperparameters"? (Check all that apply.)
 - **size of the hidden layers.**
 - **learning rate**
 - **number of iterations**
 - **number of layers in the neural network**
6. True/False? Vectorization allows you to compute forward propagation in an L -layer neural network without an explicit for-loop (or any other explicit iterative loop) over the layers $l = 1, 2, \dots, L$.
 - True

- **False**
7. True or false? A language model usually does not need labels annotated by humans for its pretraining.
- **True**
 - False
8. What is the order of the language modeling pipeline?
- First, the model, which handles text and returns raw predictions. The tokenizer then makes sense of these predictions and converts them back to text when needed.
 - First, the tokenizer, which handles text and returns IDs. The model handles these IDs and outputs a prediction, which can be some text.
 - **The tokenizer handles text and returns IDs. The model handles these IDs and outputs a prediction. The tokenizer can then be used once again to convert these predictions back to some text.**

Short answer questions

1. Look at the definition of [Stochastic] Gradient Descent in PyTorch:
<https://pytorch.org/docs/stable/generated/torch.optim.SGD.html>. You will see that this is a bit more complex than what we have seen in the class. Let's understand a few nuances here.
 - a. Notice the `maximize` parameter which controls whether we are running a maximization or minimization. In the algorithm the effect of this parameter is shown as essentially a change in the sign of the gradients:

if *maximize*

$$\theta_t \leftarrow \theta_{t-1} + \gamma g_t$$

else

$$\theta_t \leftarrow \theta_{t-1} - \gamma g_t$$

How do you think this change leads to the desired outcome (maximization vs minimization)?

In normal stochastic gradient descent, parameters are updated by subtracting the gradient, and this moves them in the direction of steepest decrease and minimizes the objective function. When the *maximize*

parameter changes the sign of the gradient, the update instead adds the gradient to the parameters, which causes them to move in the direction of steepest increase. This turns gradient descent into gradient ascent. Flipping the sign of the gradient switches the optimization process from minimizing the objective to maximizing it.

- b. The next set of parameters `momentum`, `dampening`, `weight_decay`. What do you think the impact of these parameters are? Read the documentations and interpret their roles.

The `momentum` parameter accelerates optimization by accumulating a running average of past gradients so that the updating moves in a consistent direction. `Dampening` controls how much the current gradient contributes to the momentum term, with higher dampening reducing the influence of the new gradient and weakening the momentum effect. `Weight_decay` implements L2 regularization by adding a penalty proportional to the parameter values to the gradient update. This regularizes the process and discourages large weights and helps prevent overfitting.

2. What are few benefits to using Fixed-Window-LM over n -grams language models? (limit your answer to less than 5 sentences).

Fixed-Window language models use distributed word embeddings that allow them to generalize across similar words instead of having to treat each word as an independent symbol like n-gram models do. They reduce sparsity because they do not rely on exact count matching of word sequences. They require fewer parameters than high-order n-gram models, whose parameter count can grow exponentially with vocabulary size, and they can capture smoother patterns and better learn representations by using shared weights.

3. When searching over hyperparameters, a typical recommendation is to do random search rather than a systematic grid search. Why do you think that is the case? (less than 2 sentences).

Random search is often better than grid search because usually only a few hyperparameters matter most to model performance, and random sampling explores more distinct values of those important dimensions than grid search. A poorly defined grid search can waste trials evaluating unimportant hyperparameter combinations.

4. Remember the normalization layer that we saw during the class. Here is the corresponding PyTorch page:
<https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html>. In the normalization formula, why do we use epsilon ϵ in the denominator?

In PyTorch's LayerNorm, epsilon ϵ is added to the variance term in the denominator for numerical stability. It prevents division by zero or very small variance values, which could cause extremely large activations or unstable gradients because of floating-point precision limits.

Softmax Smackdown: Squishing the Competition

Softmax gradient

You might remember in the midterm exam that we saw a neural network with a Softmax and a cross-entropy loss:

$$\hat{y} = \text{Softmax}(h), J = \text{CE}(y, \hat{y}), \quad h, y, \hat{y} \in \mathbb{R}^d.$$

Basically here \hat{y} is a d -dimensional probability distribution over d possible options (i.e. $\sum_i y_i = 1$ and $\forall i: y_i \in [0,1]$). y is a d -dimensional one-hot vector, i.e., all of these values are zeros except one that corresponds to the correct label.

Here we want to prove the hint that was provided in the exam, which was:

$$\nabla_h J = \hat{y} - y.$$

Prove the above statement. In your proof, use the statement that you proved in HW3 about gradient of Softmax function: $\frac{d\hat{y}_i}{dh_j} = \hat{y}_i (\delta_{ij} - \hat{y}_j)$, where δ_{ij} is the Kronecker delta function and \hat{y}_i is the value in the i -th index of \hat{y} .

Let cross-entropy loss function $J = -\sum_{i=1}^d y_i \log(\hat{y}_i)$

We can compute the gradient w.r.t h_j : $\frac{\partial J}{\partial h_j} = -\sum_{i=1}^d y_i \left(\frac{1}{\hat{y}_i} \right) \left(\frac{\partial \hat{y}_i}{\partial h_j} \right)$

Using the given statement $\frac{d\hat{y}_i}{dh_j} = \hat{y}_i (\delta_{ij} - \hat{y}_j)$ we can say:

$$\frac{\partial J}{\partial h_j} = -\sum_{i=1}^d y_i \left(\frac{1}{\hat{y}_i} \right) \hat{y}_i (\delta_{ij} - \hat{y}_j) = -\sum_{i=1}^d y_i (\delta_{ij} - \hat{y}_j) = -\sum_{i=1}^d y_i \delta_{ij} + \sum_{i=1}^d y_i \hat{y}_j$$

And since y is one-hot, we know: $\sum_i y_i \delta_{ij} = y_j, \sum_i y_i = 1$

And thus: $\frac{\partial J}{\partial h_j} = -y_j + \hat{y}_j = \hat{y}_j - y_j$

Since this holds for every j , we have: $\nabla_h J = \hat{y} - y$

Softmax temperature

Remember the softmax function, $\sigma(\mathbf{z})$? Here we will add a *temperature* parameter $\tau \in \mathbb{R}^+$ to this function:

$$\text{Softmax: } \sigma(\mathbf{z}; \tau)_i = \frac{e^{z_i/\tau}}{\sum_{j=1}^K e^{z_j/\tau}} \quad \text{for } i = 1, \dots, K$$

Show the following:

1. In the limit as temperature goes to zero $\tau \rightarrow 0$, softmax becomes the same as greedy action selection, argmax.

Let us start by saying $m = \max_j z_j$ and assume for simplicity that the maximum is unique and occurs at index k .

We can then rewrite the softmax by factoring out $e^{m/\tau}$:

$$\sigma(\mathbf{z}; \tau)_i = \frac{e^{(z_i - m)/\tau}}{\sum_{j=1}^K e^{(z_j - m)/\tau}}$$

For $i = k$, $z_k = m$, so $e^{\frac{z_k - m}{\tau}} = e^0 = 1$

For $i \neq k$, $z_i - m < 0$

As $\tau \rightarrow 0^+$, $\frac{z_i - m}{\tau} \rightarrow -\infty$, so $e^{(z_i - m)/\tau} \rightarrow 0$

Thus, in the denominator of the function, $\sum_{j=1}^K e^{(z_j - m)/\tau} \rightarrow 1$

Therefore, for $i = k$, $\sigma(z; \tau)k \rightarrow 1$ and for $i \neq k$, $\sigma(z; \tau)k \rightarrow 0$

So as $\tau \rightarrow 0$, the softmax distribution converges to a one-hot vector placing all probability mass on the index $k = \operatorname{argmax}_j z_j$.

2. In the limit as temperature goes to infinity $\tau \rightarrow +\infty$, softmax gives equiprobable selection among all actions.

First, we can consider the limit $\tau \rightarrow +\infty$. Then for every i we have $\frac{z_i}{\tau} \rightarrow 0$

Using the first-order Taylor expansion of the exponential around 0 we can say:

$$e^{\left(\frac{z_i}{\tau}\right)} \approx 1 + \frac{z_i}{\tau}$$

As $\tau \rightarrow \infty$, the term $\frac{z_i}{\tau} \rightarrow 0$, so $e^{\frac{z_i}{\tau}} \rightarrow 1$

Therefore, the numerator approaches 1 for every i , and the denominator becomes:

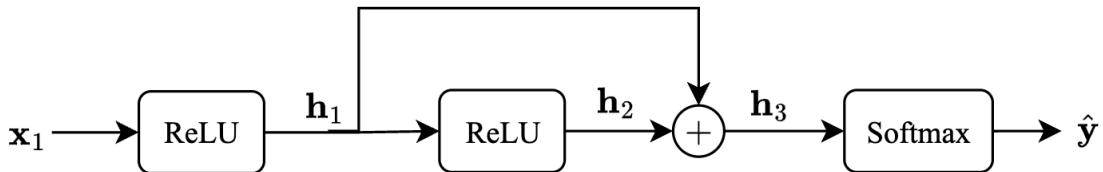
$$\sum_{j=1}^K e^{\frac{z_j}{\tau}} \rightarrow \sum_{j=1}^K 1 = K$$

And thus $\sigma(z; \tau)_i \rightarrow \frac{1}{K}$

So in the infinite-temperature limit, softmax assigns equal probability to all K actions, resulting in equiprobable (uniform) selection.

Backprop Through Residual Connections

As you know, When neural networks become very deep (i.e. have many layers), they become difficult to train due to the vanishing gradient problem – as the gradient is back-propagated through many layers, repeated multiplication can make the gradient extremely small, so that performance plateaus or even degrades. An effective approach is to add skip connections that skip one or more layers. See the provided network.



$$\mathbf{x} \in \mathbb{R}^d, \mathbf{W}_{1,2} \in \mathbb{R}^{d \times d}, \hat{\mathbf{y}} \in \mathbb{R}^d$$

$$\mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}_1, \mathbf{h}_1 = \text{ReLU}(\mathbf{z}_1),$$

$$\mathbf{z}_2 = \mathbf{W}_2 \mathbf{h}_1, \mathbf{h}_2 = \text{ReLU}(\mathbf{z}_2),$$

$$\mathbf{h}_3 = \mathbf{h}_1 + \mathbf{h}_2, \hat{\mathbf{y}} = \text{Softmax}(\mathbf{h}_3), J = \text{CE}(\mathbf{y}, \hat{\mathbf{y}})$$

1. You have seen this in the quiz, but lets try again. Prove that: $\frac{\partial}{\partial z_i} \text{ReLU}(z) = 1\{z_i > 0\}$

where $1\{x > 0\} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$. More generally, $\nabla_z \text{ReLU}(z) = \text{diag}(1\{z > 0\})$ where $1\{\cdot\}$ is applied per each dimension and $\text{diag}(\cdot)$ turns a vector into a diagonal matrix.

Given $\text{ReLU}(z_i) = \max(0, z_i)$, we consider a single z_i .

If $z_i > 0$, then $\text{ReLU}(z_i) = z_i$, so $\frac{\partial}{\partial z_i} \text{ReLU}(z_i) = 1$.

If $z_i < 0$, then $\text{ReLU}(z_i) = 0$, so $\frac{\partial}{\partial z_i} \text{ReLU}(z_i) = 0$.

At $z_i = 0$, the function is not differentiable, though we define the derivative to be 0 in practice.

Now, let $z \in \mathbb{R}^d$ and define ReLU elementwise: $\text{ReLU}(z) = \begin{bmatrix} \text{ReLU}(z_1) \\ \vdots \\ \text{ReLU}(z_d) \end{bmatrix}$

Then, each output coordinate depends only on its corresponding input coordinate.

$$\text{So, } \frac{\partial \text{ReLU}(z_i)}{\partial z_j} = \begin{cases} 1\{z_i > 0\}, i = j \\ 0, i \neq j \end{cases}$$

And this means the Jacobian matrix is diagonal, with entries $1\{z_i > 0\}$ on the diagonal. Thus:

$$\nabla_z \text{ReLU}(z) = \text{diag}(1\{z > 0\})$$

2. As you see, a single variable (\mathbf{h}_2 in the example) feeds into two different layers of the network. Here we want to show that the two upstream signals stemming from this node are merged as summation during Backprop. Specifically prove that:

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_1} = I + \frac{\partial \mathbf{h}_2}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{h}_1}$$

We start by differentiating h_3 :

Since $h_3 = h_1 + h_2$, taking the derivative w.r.t. h_1 :

$$\frac{\partial h_3}{\partial h_1} = \frac{\partial h_1}{\partial h_1} + \frac{\partial h_2}{\partial h_1}, \text{ where the first term is the identity matrix } I.$$

Now, because h_2 depends on h_1 through z_2 , we can apply the chain rule to say:

$$\frac{\partial h_2}{\partial h_1} = \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1}$$

And combining these, we get $\frac{\partial h_3}{\partial h_1} = I + \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1}$

3. In the given neural network your task is to **compute the gradient** $\frac{\partial J}{\partial x}$. You are allowed (and highly encouraged) to use variables to represent intermediate gradients.

Hint 1: Compute these gradients in order to build up your answer: $\frac{\partial J}{\partial h_3}, \frac{\partial J}{\partial h_2}, \frac{\partial J}{\partial z_2}, \frac{\partial J}{\partial h_1}, \frac{\partial J}{\partial z_1}, \frac{\partial J}{\partial x}$. Show your work so we are able to give partial credit!

Hint 2: Recall that *downstream = upstream * local*.

First, for softmax and cross entropy we have: $\frac{\partial J}{\partial h_3} = \hat{y} - y$

Let $\delta_3 := \frac{\partial J}{\partial h_3} = \hat{y} - y$

Since $h_3 = h_1 + h_2$, $\frac{\partial h_3}{\partial h_2} = I$ and thus $\frac{\partial J}{\partial h_2} = \delta_3$

Then, using the ReLU derivative: $\frac{\partial h_2}{\partial z_2} = \text{diag}(1\{z_2 > 0\})$

thus $\frac{\partial J}{\partial z_2} = \delta_3 \odot 1\{z_2 > 0\}$

and let $\delta_2 := \frac{\partial J}{\partial z_2}$

Now, for the gradient w.r.t h_1 , 2 paths contribute to h_1

For the path that skips the connection, $\left. \frac{\partial J}{\partial h_1} \right|_{\text{direct}} = \delta_3$

and for the path through layer 2: $\left. \frac{\partial J}{\partial h_1} \right|_{\text{via } z_2} = W_2^T \delta_2$

and let $\delta_1 := \frac{\partial J}{\partial h_1}$

Using ReLU again, we can say $\frac{\partial J}{\partial z_1} = \delta_1 \odot 1\{z_1 > 0\}$, let $\delta_0 := \frac{\partial J}{\partial z_1}$

Since $z_1 = W_1 x$, $\frac{\partial z_1}{\partial x} = W_1$, thus $\frac{\partial J}{\partial x} = W_1^T \delta_0$

So, our final answer is $\frac{\partial J}{\partial x} = W_1^T [(\delta_3 + W_2^T \delta_2) \odot 1\{z_1 > 0\}]$

4. Based on your derivations, explain why residual connections help mitigate vanishing gradients.

From our derivations, the gradient w.r.t h_1 is $\delta_3 + W_2^T \delta_2$.

The important piece is the direct additive term δ_3 coming from the skip connection. Even if the second path (through W_2 and ReLU) produces very small gradients due to small weights or saturated activations, the identity path still passes the upstream gradient directly backward without being multiplied by additional weight matrices or nonlinear derivatives. This is proof that residual connections can create a shortcut for gradients and help them flow backward with minimal attenuation.

Programming

In this programming homework, we will

- implement your own subword tokenizer.
- implement fixed-window MLP language models

Skeleton Code and Structure:

The code base for this homework can be found at [this GitHub repo](#) under the hw3 directory. Your task is to fill in the missing parts in the skeleton code, following the requirements, guidance, and tips provided in this pdf and the comments in the corresponding .py files. The code base has the following structure:

- `tokenization.py` implements the Byte-Pair Encoding algorithm for subword tokenization.
- `mlp_lm.py` implements a fixed-window MLP-based language model on a subset of Wikipedia.
- `main.py` provides the entry point to run your implementations in both `tokenizatiton.py` and `mlp_lm.py`.
- `hw4.md` provides instructions on how to setup the environment and run each part of the homework in `main.py`

TODOs — Your tasks include 1) generate plots and/or write short answers based on the results of running the code; 2) fill in the blanks in the skeleton to complete the code. We will explicitly mark these plotting, written answer, and filling-in-the-blank tasks as **TODOs** in the following descriptions, as well as a `# TODO` at the corresponding blank in the code.

Submission:

Your submission should contain two parts: 1) plots and short answers under the corresponding questions below; and 2) your completion of the skeleton code base, in a `.zip` file

Tokenization Algorithms

All NLP systems have *at least* three main components that help machines understand natural language:

1. Tokenization
2. Embedding

3. Model architectures

Models like BERT, GPT-2 or GPT-3 all share the same components but with different architectures that distinguish one model from another. We are going to focus on the first component of an NLP pipeline which is **tokenization**. An often overlooked concept but it is a field of research in itself. Though we have SOTA algorithms for tokenization it's always a good practice to understand the evolution trail.

Here's what we'll cover:

- What is tokenization?
- Why do we need a tokenizer?
- Types of tokenization - Word, Character and Subword.
- Byte Pair Encoding Algorithm
- Other tokenization algorithms:
 - Unigram Algorithm
 - WordPiece - BERT transformer
 - SentencePiece - End-to-End tokenizer system

What is Tokenization?

Tokenization is the process of representing raw text in smaller units called tokens. These tokens can then be mapped with numbers to further feed to an NLP model.

Read and run the `full_word_tokenization_demo` in `tokenization.py` for an overly simplified example of what a tokenizer does.

This is a very simple example where we just split a text string in to “whole words” on white spaces, and we have not considered grammar, punctuation, or compound words (like “test”, “test-ify”, “test-ing”, etc.).

Problems with word tokenization:

- **Missing words in the training data:** With word tokens, your model won't recognize the variants of words that were not part of the data on which the model was trained. So, if your model has seen ‘foot’ and ‘ball’ in the training data but the final text has “football”, the model won't be able to recognize the word and it will be treated with a “<UNK>” token. Similarly, punctuations pose another problem, “let” or “let’s” will need individual tokens and it is an inefficient solution. This will **require a huge vocabulary** to make sure you've every variant of the word. Even if you add a

lemmatizer to solve this problem, you're adding an extra step in your processing pipeline.

- **Not all languages use space for separating words:** For a language like Chinese, which doesn't use spaces for word separation, this tokenizer will fail.

Character-based tokenization

To resolve the problems associated with word-based tokenization, an alternative approach of character-by-character tokenization was tried. This solves the problem of missing words as now we are dealing with characters that can be encoded using ASCII or Unicode and it could generate embedding for any word now.

Every character, be it space, apostrophes, or colons, can now be assigned a symbol to generate a sequence of vectors. But this approach had its cons. Character-based models will treat each character and will lead to longer sequences. For a 5-word long sentence, you may need to process 30 tokens instead of 5 word-tokens.

Subword Tokenization

To address the earlier issues, we could think of breaking down the words based on a set of prefixes and suffixes. For example, we can build a system that can identify subwords like “##s”, “##ing”, “##ify”, “un##” etc., where the position of double hash “##” denotes prefix and suffixes. So, a word like “unhappily” is tokenized using subwords like “un##”, “happ”, and “##ily”.

BPE (Byte-Pair Encoding) was originally a data compression algorithm that is used to find the best way to represent data by identifying the common byte pairs. It is now used in NLP to find the best representation of text using the least number of tokens.

Here's how it works:

1. Add a “</w>” at the end of each word to identify the end of a word and then calculate the word frequency in the text.
2. Split the word into characters and then calculate the character frequency.
3. For a predefined number of iterations (set by you), count the frequency of the consecutive tokens and merge the most frequently occurring pairings.
4. Keep iterating until you have reached the iteration limit or if you have reached the token limit.

We will now go through this algorithm step by step.

Read the `get_word_freq` function in `tokenization.py` that implements the step 1 above to preprocess each word and count its frequency.

TODOs: Read and complete the missing lines in the `get_pairs` function in `tokenization.py` to implement the step 2 above. This function takes the word frequency

record returned from `get_word_freq` as input, split the words into tokens (characters at this initial step), and counts token-pairs frequency.

Hint: follow the comments in the code for specific requirements.

Read the `get_most_frequent_pair` and `merge_byte_pairs` functions in `tokenization.py` that implements the step 3 above, merge the top-1 frequent token-pairs in all the words.

In practise, after iterating over the input text for a desired number of times, we extract the resulting tokenization as our subword dictionary. For this purpose, we provide `get_subword_tokens` function in `tokenization.py`

With all these functions, we can now run one step of the BPE algorithm!

Read and run the `test_one_step_bpe.py` in `main.py` to test the BPE algorithm for one step. A correct implementation of `get_pairs` should pass all the tests.

Complete the BPE Algorithm

With the above implementations, we can complete all the steps of the BPE algorithm that iterates over the input corpus.

TODOs: Read and complete the missing lines in the `extract_bpe_subwords` function in `tokenization.py` that implements the full BPE algorithm. Once you finished, run the `test_bpe` function in `main.py`, a correct implementation of `extract_bpe_subwords` should pass all the tests.

Hint: follow the comments in the code and you can reuse the helper functions provided/you implemented above.

So as we iterate with each best pair, we merge (concatenating) the pair and you can see as we recalculate the frequency, the original character token frequency is reduced and the new paired token frequency pops up in the token dictionary.

Running BPE on a subset of Wikipedia

Now, let's run this algorithm on a larger scale. In particular, we will run on many Wikipedia paragraphs. As usual, let's use the Huggingface library to download the data (the `load_bpe_data` function).

TODOs: Read and run the `bpe_on_wikitext` in `main.py` that iterates the BPE algorithm on a subset of Wikipedia, and interprets the final subwords extracted from the Wikipedia documents. In particular, identify examples of subwords that correspond to

1. complete words

`Vietnamese</w>, March</w>, music</w>`

2. part of a word

`contro, depend, gen, lim, ing`

3. non-English words(including other languages or common sequence special characters)

`λ, θ, μ, ν, α, δ, €, £, $, %`

Explain why these subwords have come about. (no more than 10 sentences)

The BPE algorithm shrinks the training corpus by merging frequent character sequences. Common full words are included as single tokens, but common subwords like prefixes, suffixes, and stems are merged because they appear across many different words. Rare words are broken into smaller reusable pieces to reduce vocabulary size. The non-English scripts and symbols are present because the Wikipedia data is multilingual. Special characters and punctuation appear because they occur frequently in formatted text like the Wikipedia data.

Additonal Readings: Other Tokenization Algorithms

WordPiece

WordPiece is the subword tokenization algorithm used for [BERT](#), [DistilBERT](#), and [Electra](#). The algorithm was outlined in and is very similar to BPE. WordPiece first initializes the vocabulary to include every character present in the training data and progressively learns a given number of merge rules. In contrast to BPE, WordPiece does not choose the most frequent symbol pair, but the one that maximizes the likelihood of the training data once added to the vocabulary.

So what does this mean exactly? Referring to the previous example, maximizing the likelihood of the training data is equivalent to finding the symbol pair, whose probability is divided by the probabilities of its first symbol followed by its second symbol is the greatest among all symbol pairs. E.g. “u”, followed by “g” would have only been merged if the probability of “ug” divided by “u”, “g” would have been greater than for any other symbol pair. Intuitively, WordPiece is slightly different from BPE in that it evaluates what it *loses* by merging two symbols to ensure it’s *worth it*.

Unigram

Unigram is a subword tokenization algorithm introduced in . In contrast to BPE or WordPiece, Unigram initializes its base vocabulary to a large number of symbols and progressively trims down each symbol to obtain a smaller vocabulary. The base vocabulary could for instance correspond to all pre-tokenized words and the most common substrings. Unigram is not used directly for any of the models in the transformers, but it’s used in conjunction with SentencePiece([\[paragraph:sentencetpiece\]](#)).

At each training step, the Unigram algorithm defines a loss (often defined as the log-likelihood) over the training data given the current vocabulary and a unigram language model. Then, for each symbol in the vocabulary, the algorithm computes how much the overall loss would increase if the symbol was to be removed from the vocabulary. Unigram then removes p (with p usually being 10% or 20%) percent of the symbols whose loss

increase is the lowest, i.e. those symbols that least affect the overall loss over the training data. This process is repeated until the vocabulary has reached the desired size. The Unigram algorithm always keeps the base characters so that any word can be tokenized.

Because Unigram is not based on merge rules (in contrast to BPE and WordPiece), the algorithm has several ways of tokenizing new text after training. As an example, if a trained Unigram tokenizer exhibits the vocabulary:

[“b”, “g”, “h”, “n”, “p”, “s”, “u”, “ug”, “un”, “hug”],

“hugs” could be tokenized both as [“hug”, “s”], [“h”, “ug”, “s”] or [“h”, “u”, “g”, “s”]. So which one to choose? Unigram saves the probability of each token in the training corpus on top of saving the vocabulary so that the probability of each possible tokenization can be computed after training. The algorithm simply picks the most likely tokenization in practice but also offers the possibility to sample a possible tokenization according to their probabilities.

Those probabilities are defined by the loss the tokenizer is trained on. Assuming that the training data consists of the words x_1, \dots, x_N and that the set of all possible tokenizations for a word x_i is defined as $S(x_i)$, then the overall loss is defined as

$$\mathcal{L} = - \sum_{i=1}^N \log \left(\sum_{x \in S(x_i)} p(x) \right)$$

SentencePiece

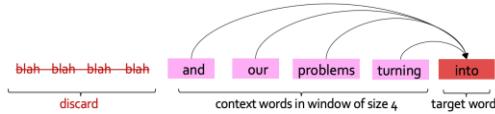
All tokenization algorithms described so far have the same problem: It is assumed that the input text uses spaces to separate words. However, not all languages use spaces to separate words. One possible solution is to use language-specific pre-tokenizers, e.g. [XLM](#) uses a specific Chinese, Japanese, and Thai pre-tokenizer). To solve this problem more generally, treats the input as a raw input stream, thus including the space in the set of characters to use. It then uses the BPE or unigram algorithm to construct the appropriate vocabulary.

The [XLNetTokenizer](#) uses SentencePiece for example, which is also why in the example earlier the “” character was included in the vocabulary. Decoding with SentencePiece is very easy since all tokens can just be concatenated and “” is replaced by a space.

All transformer models in the library that use SentencePiece use it in combination with unigram. Examples of models using SentencePiece are [ALBERT](#), [XLNet](#), [Marian](#), and [T5](#).

Fixed-Window MLP Language Models

In the second part of the homework, we will build, train, and evaluate fixed-window MLP language models as we learned in class: given the context words in the fixed-size window on the left hand side, predict the next word continuation.



Fixed-window LM

Data Loading and Preprocessing

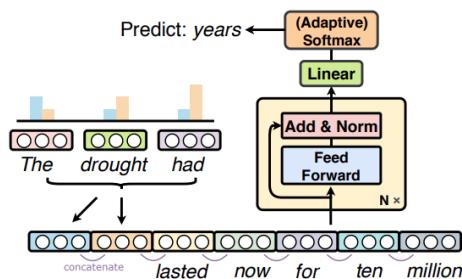
We will be using the same [WikiText](#) dataset we used for building n-gram LM in homework 2. We follow similar preprocessing steps to split paragraphs into sentences followed by tokenization (Now we have learned more about subword tokenization in class and the previous question!).

TODOS read the `preprocess_data` function in `ngram_lm.py` and complete the missing lines to prepare input-output pairs for training fixed-window LMs.

Hint: follow the requirements in the code comments.

Build our LM

Let's now turn to building our model. Here, we are following the footsteps of the neural probabilistic language model (NPLM) , which is extending earlier studies such as .



NPLM Architecture

We will follow a modular design for our implementation to make it more interpretable. In particular, will implement 3 layers:

- The **input layer** which loops up word embeddings, concatenates them, and transforms them into a hidden representation.
- The **middle layer** transforms the representation with a non-linearity.
- The **output layer** which transforms a hidden vector to a probability distribution over the words

Let's get to work!

TODOS: read and complete the forward functions for the following classes

- `NPLM_first_block`: input layer that embeds the input token ids into embedding vectors, concatenates the embeddings, applies linear transformation, layer normalization, and dropout
- `NPLM_block`: middle layers with linear transformation, tanh activation, residual connection, layer normalization, and dropout.
- `NPLM_final_block`: output layer that transforms hidden representation to log probability over the vocabulary with log softmax.

Hint: check out `nn.LayerNorm` and `nn.Dropout`, `torch.tanh` and `nn.functional.log_softmax` for more details on these layers and operations. For embedding concatenation, you may find `torch.Tensor.view` useful. Also, follow the step-by-step comments in the code for the requirements of the implementation.

TODOs: read and complete the `__init__` and `forward` functions for the `NPLM` class, which is the final model that stacks all the above layers.

Hint: remember to apply ReLU non-linear activation after each middle layer, details at `nn.functional.relu`.

Train and Evaluate the LM

Now it's time to train and evaluate it! Like the previous homework, we will use cross-entropy loss between the predictions of our language model and actual words that appeared in our training data. Note that we applied log softmax to the final output of the LM, as described in homework 2, we will use `negative log-likelihood loss` as the training criterion to calculate the cross-entropy loss.

As introduced in the class, we will use **Perplexity** to evaluate our LM, as a measure of predictive quality of a language model.

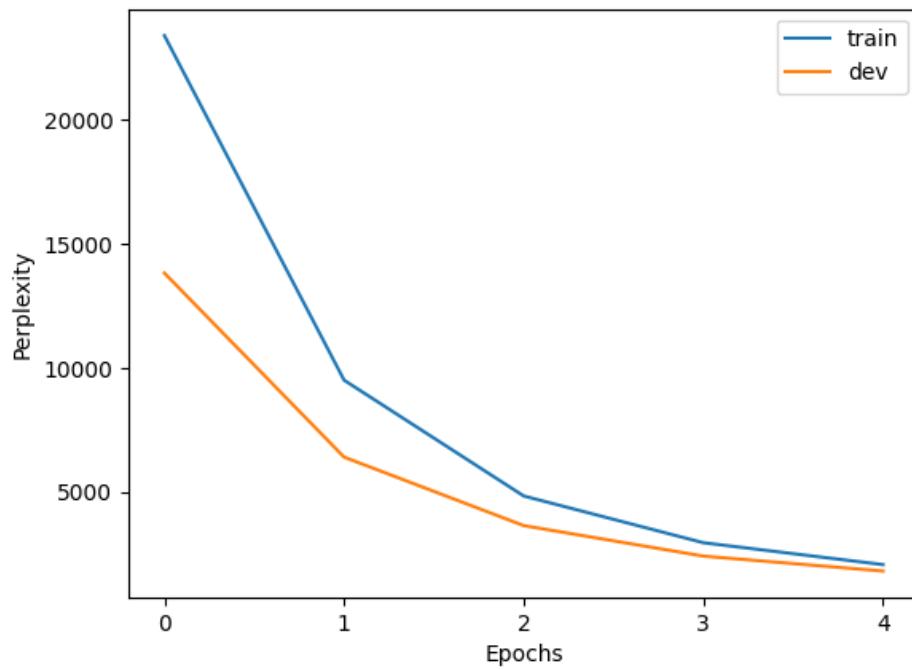
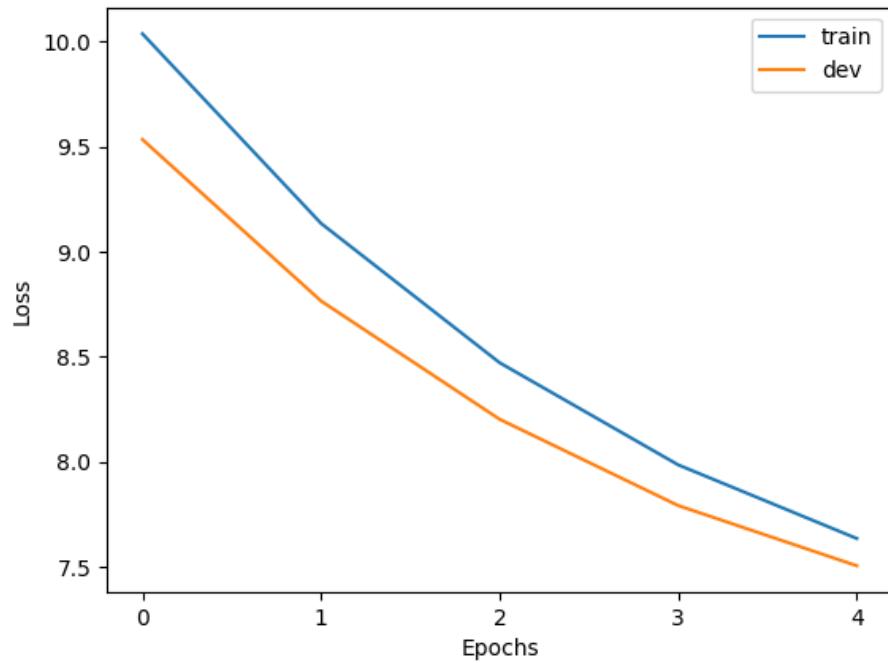
TODOs Read and complete the missing lines in `train` and `evaluate` functions in `mlp_lm.py` to calculate the perplexity.

Hint: remember in the class we discussed the connection between perplexity and cross-entropy loss, and note that in our implementation we took natural logarithm instead of the base of 2.

Note we are using Adam (**Adaptive Moment Estimation**) , which is a popular optimization algorithm used in deep learning and machine learning. It is a stochastic gradient descent (SGD) optimization algorithm that is well suited for training deep neural networks. The algorithm has some internal estimates to dynamically adjust the learning rates for each parameter based on its past gradients, which can result in faster convergence and improved performance compared to traditional SGD.

TODOs Once you finished all the implementations, run `load_data_mlp_lm` and

`single_run_mlp_lm` in `main.py` to train and evaluate the model and paste the plots of loss and Perplexity here.



Sample From a Pre-trained LM

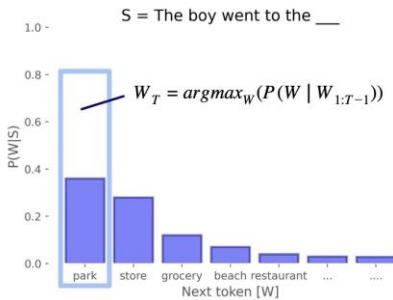
Note that compared with the original NPLM implementation, we reduce the model size by shrinking both the width and depth of our model, and we only use a small subset of the data. It is expected to take an hour or so to train the model on the CPU of your PC. Training the above LM with full size model and data might take hours to days. To save you time, we have trained a language model for you to play with. All you have to do is to download its weight parameters. Download the [pre-trained weights](#) and copy it to your local directory under /hw4/.

Now that we have the model weight downloaded, we can instantiate a model with these parameters. This will work in two steps.

- First we will need to create a new model (with potentially random weights).
- Then, we will copy the parameter values to the model using [load_state_dict](#) function.

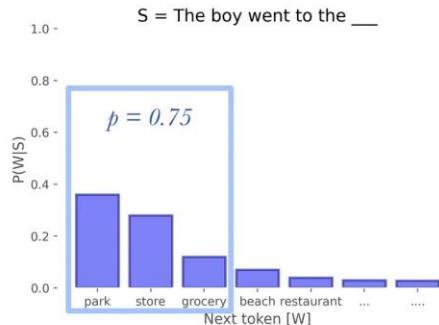
We then evaluate the loaded model on the dev set. Now that we have loaded a pre-trained LM, how can we sample from it? We will implement two strategies, greedy decoding and top-p sampling that we briefly discussed in homework 2.

For greedy, we select the most probable words (argmax).



Greedy Decoding

For top-p, we sample from the distribution proportional to word probabilities. Words with higher probabilities will be more likely to be sampled. However, we also have an option of filtering the low-probability tokens, and instead retaining tokens that constitute top_p probability.



Greedy Decoding

Read `generate_text` and `sample_from_mlp_lm` in `mlp_lm.py` for more details about how to load the pre-trained model, evaluate on dev set and perform greedy/top-p sampling. You can learn more about top-p sampling implementation at [TopPLogitsWarper](#).

TODOs Run the `sample_from_trained_mlp_lm` in `main.py` to sample from the pre-trained LM, paste the completion to the prefix under different sampling strategies here, and describe in 2-3 sentences your findings.

Hint: compare the output of the above generations, which one is your favorite? Explain why this choice of sampling leads to better text generation.

----- Sample From the Model -----

----- greedy -----

Generated text: The best perks of living on the east side of the city. ". "...".... ". "...".... ". "

----- sampling with p=0.0 -----

Generated text: The best perks of living on the east side of the city. ". "...".... ". "...".... ". "

----- sampling with p=0.3 -----

Generated text: The best perks of living on the east of the U. S. government had a year. ". ", the episode ' s very much of the most common age. "...

----- sampling with p=1.0 -----

Generated text: The best perks of living on the east slopes of Mexico and lip Germany was once over the 20th century but none of Rose concluded children, adopted with the frame of the magnificent Age by trees.

For greedy decoding and $p = 0.0$, the outputs are the same, which makes sense because the model is always picking the single most likely next token. For $p = 0.3$, the text is more

varied but still awkward and disconnected, as the model is reaching for slightly less probable words without fully maintaining coherence. With $p = 1.0$, the model has the freedom to sample from the entire distribution, which makes the writing more creative but also much less sensible.

Optional Feedback

Have feedback for this assignment? Found something confusing? We'd love to hear from you!